Mixed-Size Concurrency: ARM, POWER, C/C++11, and SC

Shaked Flur[1] Susmit Sarkar[2] Christopher Pulte[1] Kyndylan Nienhuis[1] Luc Maranget[3]
Kathryn E. Gray[1] Ali Sezgin[1] Mark Batty[4] Peter Sewell[1]

[1] University of Cambridge, UK first.last@cl.cam.ac.uk
[2] University of St Andrews, UK ss265@st-andrews.ac.uk
[3] INRIA, France luc.maranget@inria.fr
[4] University of Kent, UK M.J.Batty@kent.ac.uk

Abstract

Previous work on the semantics of relaxed shared-memory concurrency has only considered the case in which each load reads the data of exactly one store. In practice, however, multiprocessors support mixed-size accesses, and these are used by systems software and (to some degree) exposed at the C/C++ language level. A semantic foundation for software, therefore, has to address them.

We investigate the mixed-size behaviour of ARMv8 and IBM POWER architectures and implementations: by experiment, by developing semantic models, by testing the correspondence between these, and by discussion with ARM and IBM staff. This turns out to be surprisingly subtle, and on the way we have to revisit the fundamental concepts of coherence and sequential consistency, which change in this setting. In particular, we show that adding a memory barrier between each instruction does not restore sequential consistency. We go on to also extend the C/C++11 model to support non-atomic mixed-size memory accesses.

This is a necessary step towards semantics for real-world shared-memory concurrent code, beyond litmus tests.

Categories and Subject Descriptors C.0 [*General*]: Modeling of computer architecture; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Relaxed Memory Models, mixed-size, semantics, ISA

1. Introduction

The shared-memory abstractions provided by multiprocessors are relaxed: to accommodate a range of hardware optimisations, they provide weaker guarantees than the sequential consistency model (articulated by Lamport [1]), in which the writes and reads of any execution can be totally ordered, with reads reading from the most recent writes. Relaxed memory hardware dates back at least to the mid-1970s and it is now ubiquitous, e.g. in the ARM, IBM POWER, Itanium, MIPS, Sparc, and x86 architectures. This has prompted much research into the semantics that multiprocessors could or actually do provide, including [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40]. Recent work among this has established semantic models for x86 [32], IBM POWER [33, 34, 35, 36, 38], and ARM [39] that are validated both by experiment against multiprocessor implementations and by discussion with the vendor architects.

All this previous work, however, makes the simplifying assumption that memory accesses are to some abstract notion of store locations, or, equivalently, that all accesses are of the same size and are suitably aligned (the only substantive exception we are aware of is the Itanium specification [22]). In reality, all these architectures support accesses at multiple sizes (typically at least 1, 2, 4, and 8-byte units). Code routinely uses all these, and also routinely accesses the same memory with mixed-size accesses. For a simple but common example, C structures may be copied using `memcpy`, which accesses their members byte-by-byte (or in larger units in optimised implementations) irrespective of their natural sizes. Moreover, C compilers, while normally allocating memory in aligned units, typically also support packed structs, in which the members are adjacent and therefore potentially misaligned. One would hope that these idioms normally occur only in sequential or non-racy code, but concurrent algorithms and other systems code also make essential use of mixed-size accesses, including, for example, the Linux kernel `lockref` implementation, ARMv8 ticketed spinlock implementation, and read-copy-update (RCU) code, and the FreeBSD/i386 manipulation of PAE page table bits. Looking at the first in more detail, the Linux kernel `lockref` [41] combines a spinlock and a reference count. It is defined in `lockref.h` as a union of an 8-byte whole and two 4-byte structure members:

```
struct lockref {
  union {
    aligned_u64 lock_count;  // whole lockref
    struct {
      spinlock_t lock;       // lock
      int count;             // reference count
    };
  };
};
```

This lets fastpath code update the reference count with a relaxed 8-byte compare-exchange, without taking the lock, with non-fastpath code first taking the lock and then updating the reference count, using 4-byte accesses.

All this is beyond the scope of previous relaxed-memory semantic models: they do not cover even simple non-racy mixed size cases, let alone these more intricate concurrent examples; they suffice for litmus tests, but not for typical real code.

In this paper we develop semantic models that cover the mixed-size behaviour of ARM and IBM POWER, building on previous work for the non-mixed-size case ([39] and [33, 34, 35, 36, 38] respectively). To the best of our knowledge, our models exactly capture the architectural intent for each, for the ISA fragments we

deal with: the envelope of behaviours intended to be allowed. We develop the models and establish confidence in them by an iterative process, following [28, 31, 30, 33, 39]:

1. referring to the architecture texts [42, 43], where those are clear;
2. experimental testing of POWER and ARM processor implementations, with handwritten litmus tests that explore key questions, using the `litmus` tool [44], that we have extended to support mixed-size litmus tests;
3. detailed discussion with IBM and ARM architects about their architectural intent, our experimental results, and the structure of our models;
4. expressing the models in rigorous, unambiguous mathematics, which itself identifies corner cases;
5. generating executable code from the models that can calculate the set of all allowed behaviours of small litmus tests, for comparison with experimental data from running those tests on hardware implementations, and allow interactive exploration of the model behaviour, in command-line and web interfaces.

We describe the mixed-size phenomena of the architectures and hardware implementations in §2, and our models in §3. Some of this is intricate but essentially straightforward, e.g. the splitting of misaligned accesses into atomic parts, but handling accesses with distinct but overlapping footprints turned out to have surprisingly subtle consequences for the fundamental notion of coherence, and for the semantics of barriers; it also interacts delicately with write forwarding. Our experimental testing also identified new errata in two production multiprocessor implementations, one of which involved mixed-size phenomena and was found with a test arising from our model design; these have been reported to and acknowledged by the vendors.

The conventional wisdom for most hardware memory models is that adding sufficient memory barriers, e.g. a strong barrier between each memory access, will restore sequential consistency. To the best of our knowledge this has been taken for granted for the ARM and POWER architectures, but in the mixed-size setting this supposedly fundamental property turns out not to hold, as we show in §4. We define a weaker notion, BSC+SCA, and show it characterises the behaviour of fully barriered ARM and POWER programs without misaligned accesses. We also show that mixed-size ARM programs that use only the ARM write-release and read-acquire instructions are sequentially consistent.

Turning to the C language level, there are two main cases to consider. First, there is "well-behaved" C code using the C/C++11 concurrency support [45, 46], in which shared-memory accesses are either non-atomic and should be protected (by locks or other synchronisation), or are expressed as C/C++11 *atomic* accesses, with one of the memory orders provided (SC, release/acquire, release/consume, or relaxed). In the C/C++11 model, programs that exhibit data races on non-atomics (or between non-atomic and atomic accesses) are deemed to have undefined behaviour, and the effective type rules of the ISO C standard prohibit mixed-size use of atomics. These restrictions should rule out programmer-observable instances of the hardware mixed-size phenomena that we see in §2, in accordance with an implicit design goal for relaxed-memory architectures, that for well-behaved programs the associated hardware optimisations should not be programmer-visible. For this, we define an extension of the C/C++11 axiomatic concurrency model [46] to cover mixed-size nonatomic accesses (§5), and sketch an argument that the standard compilation scheme from C/C++11 concurrency to POWER concurrency is unaffected by mixed-size phenomena (§6). This argument builds on previous proof attempts [34, 35]. It has recently become clear that those are unsound [47, 48, 49], but those issues and the mixed-size extensions appear to be orthogonal.

Second, there is the case of low-level C/C++ code that intentionally uses racy mixed-size accesses. Such programs have undefined behaviour according to the ISO standard, but there are important instances in practice, e.g. as mentioned above. For these, the clarification of the hardware behaviour that we provide in this paper is directly relevant, but programmers currently must reason about such code in terms of the assembly generated by their compiler, as there is no candidate source-language semantics that admits both the mixed-size hardware phenomena with compiler optimisations. This adds to the other outstanding open problems with giving a high-level language concurrency semantics [50, 51, 52, 53], which we do not attempt to address here: thin-air values, undefined behaviour, and general combinations of non-atomic and atomic accesses.

Returning to our hardware experimental work in §7, we have run tests on a 48-hardware-thread POWER 7 machine and on five ARMv8-architecture implementations, with SoCs and cores by several vendors. The tests include the hand-written tests of §2 and §3, tests generated by the `diy` tool, which we have extended to the mixed-size case, and non-mixed-size regression tests.

We conclude with discussion of future work in §8. The on-line supplementary material [54] includes our POWER and ARM handwritten tests, experimental data, and proofs. The web-interface version of our tool is at `www.cl.cam.ac.uk/~pes20/AArch64` and `www.cl.cam.ac.uk/~pes20/Power`.

2. Mixed-Size Phenomena in Hardware

For the hardware behaviours we discuss in this section, ARM and POWER are architecturally very similar. We illustrate with POWER versions of litmus tests, but the supplementary material contains both versions and we discuss experimental results for both. All except §2.7 are handled by our models.

2.1 Reading from Multiple Writes

The most basic phenomenon of the mixed-size setting is that writes and reads may be of different sizes, with reads potentially reading from fragments of writes and from multiple writes. For example, in the sequential case, we might have a sequence of two overlapping writes:

```
a:W x   /4 = 0x03020100  (* write 4 bytes to x  *)
b:W x+2 /1 = 0x    11    (* write 1 byte to x+2 *)
```

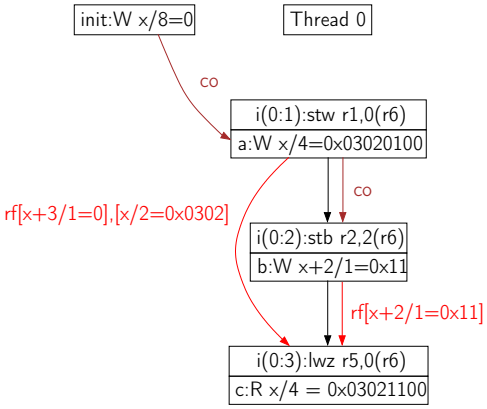
followed by a read that reads from both of them:

```
c:R x   /4 = 0x03021100  (* read 4 bytes from x *)
```

Note that this is a big-endian example. ARM and POWER both support little- and big-endian modes. In this paper we use little-endian for ARM and big-endian for POWER, following the modes used on our test machines.

In a sequential or sequentially consistent model with a concrete byte-array memory, the writes would simply update that memory in sequence, and the read would see whatever is there. In the relaxed-memory concurrent setting, we need to maintain distinct events, as shown in the execution diagram below. This shows an initialisation write (labelled `init`), the three POWER assembly instructions (the `stw`, `stb`, and `lwz`, labelled with instruction instance IDs `i(`*`tid`*`:`*`n`*`)`) and related by program-order edges (black), and the associated read and write events below each instruction (labelled `a`, `b`, `c`). In previous work such events had just an address and a value; now each event needs a *footprint*, comprising an address (e.g. `x` or `x+2`, where `x` is a pretty-printed symbol for an underlying concrete aligned 64-bit address) and a size in bytes. Also shown are the coherence relation `co` (brown) and the reads-from relation `rf` (red). Previously the `rf` relation was just a binary relation between events, but now each `rf` edge, from a write event to a read event, is labelled

with the relevant *slices* of the write: the sub-footprints of the write being read in this edge.



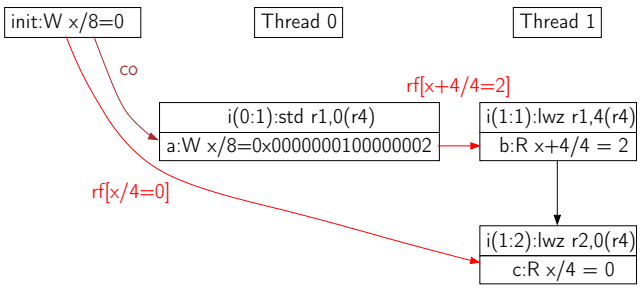
Test MIXED-SEQ-1

To fix terminology, when we say *store* or *load* we refer to assembly or machine code instructions from the ISA (instruction set architecture). When we say *write* or *read* we refer to the model events which are their main effects.

2.2 Reordering and Non-multi-copy-atomic Propagation for Disjoint Footprints

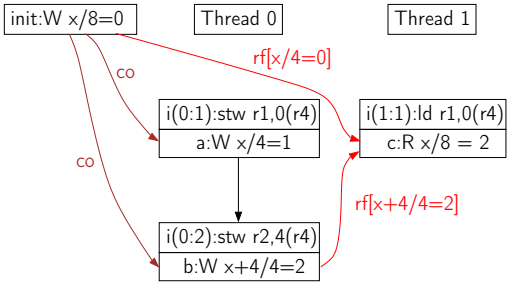
The basic choice taken by relaxed-memory architectures such as POWER and ARM is to relax program order between memory-access instructions that are not explicitly ordered in some way, to allow the hardware optimisations of unconstrained out-of-order and speculative execution (processors with stronger memory models, such as x86, may exploit similar optimisations but in more constrained ways, to ensure that they are not programmer-visible). They also allow non-multi-copy-atomic behaviour, with writes, barriers, and (for ARM) read requests allowed to propagate to other threads in multiple steps.

In the non-mixed-size setting, two instructions are "explicitly ordered" if they access the same address or they are related by some architecture-specific combinations of barriers and dependencies. In the mixed-size case, considering only aligned accesses for the moment, we have to replace that "same address" by "to overlapping footprints". This is not just a cache-line phenomenon. For example, the execution below shows a single write `a` to an 8-byte aligned `x` on Thread 1 and two reads `b` and `c` of the disjoint 4-byte footprints `x+4/4` and `x/4`. It is architecturally allowed for `b` to read from half of `a` and the program-order later `c` to read from the initial state, ignoring `a`, even though `b` and `c` are within the same cache line, and indeed within the same 64-bit footprint on a 64-bit machine. This is observable in practice (500k/3.5G instances on a POWER 7, and 6.4k/6.0G in total on our five ARMv8-architecture implementations); it is explainable by `c` being satisfied early, out-of-order.



Test CO-MIXED-1

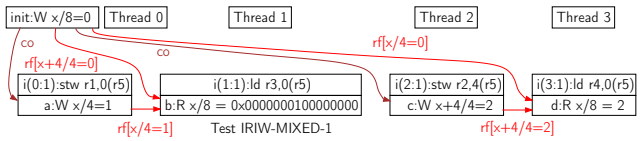
Similar behaviour is architecturally allowed for writes: the test below shows two writes to disjoint aligned 4-byte footprints on Thread 0, of which only the second is seen by the aligned 8-byte read on Thread 1. We do not observe it on current implementations — the first of several places where the vendor architectural programming models are intentionally looser than current implementations appear to be.



Test CO-MIXED-1b

The above tests are mixed-size analogues of message-passing litmus tests MP+sync+po and MP+po+addr [33], using aligned wide writes and reads in place of (respectively) the pair of two writes with a `sync` and the pair of two reads with an address dependency.

Writes to disjoint footprints also allow the non-multi-copy-atomic behaviour illustrated by IRIW, below (again using wide reads in place of the pairs of reads and address dependency of the usual IRIW+addrs). This is observed on POWER 7 (46k/1.8G) but not on our ARMv8 implementations (other non-multi-copy-atomic behaviour is likewise not observed on those, so this is not surprising).



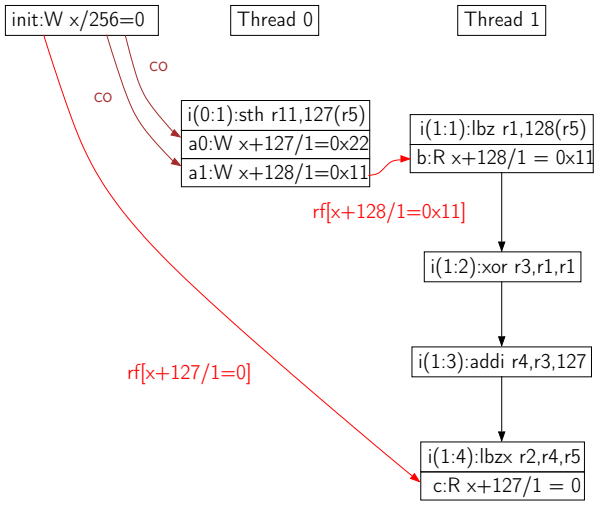
Test IRIW-MIXED-1

2.3 Atomicity of Store Instructions

In both ARM and POWER architectures, all 1, 2, 4, and 8-byte non-vector single-register accesses that are correspondingly 1, 2, 4, or 8-byte aligned are single-copy atomic[1] [43, Book II §1.4], [42, B2.6.1,B2.6.2]. Misaligned normal accesses are architecturally regarded as being split into single-byte units which are treated as independent atomic fragments, without any ordering between them.

In the previous examples all store and load instructions were aligned: we regard symbolic addresses in litmus tests (such as `x`) as maximally aligned, and the accesses were to footprints `x/8`, `x/4`, `x+4/4`, and `x+2/1`. Below, we show a non-aligned store example. Here Thread 0 writes two bytes to `x+127/2`, with a single store-half-word instruction (`sth`), and Thread 1 reads those two addresses, one at a time, with two load-byte instructions (`lbz`), first reading `x+128/1` and then `x+127/1`, with an address dependency between the two load instructions to keep them locally ordered.

[1] for POWER, the 8-byte case only for 64-bit implementations



Test MP+misaligned2+127+addr

Note that the misaligned store instruction `i(0:1)` now generates two write events (`a0` and `a1`), and the reads-from (`rf`) edges in these execution diagrams are between these individual write and read events, not between store and load instructions.

This execution is observable on POWER and ARM. The table below summarises observations for test variants with different offsets from the cache-line boundary; for each offset, we sum the results for a test as above and a variant (`MP+misaligned2+127x+addr` etc.) with loads in the opposite order.

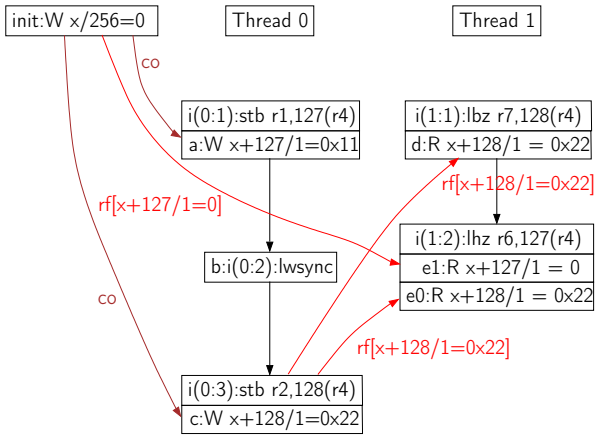
Test	Archs	POWER 7 h/w	ARMv8 h/w
`MP+misaligned2+0(x)+addr`	forbid	0/9.9G	0/9.6G
`MP+misaligned2+1(x)+addr`	allow	0/9.8G	0/9.6G
`MP+misaligned2+3(x)+addr`	allow	0/9.8G	0/9.6G
`MP+misaligned2+7(x)+addr`	allow	0/9.8G	0/9.6G
`MP+misaligned2+15(x)+addr`	allow	0/9.8G	469k/9.6G
`MP+misaligned2+31(x)+addr`	allow	4.7M/9.8G	393k/9.6G
`MP+misaligned2+63(x)+addr`	allow	4.0M/9.8G	85M/9.6G
`MP+misaligned2+127(x)+addr`	allow	12.4M/9.8G	15M/9.6G

Microarchitecturally, one would expect at least stores whose footprint spans a cache-line boundary to be split (otherwise one is in the realm of hardware transactional memory implementations, to provide atomic access to multiple cache lines while avoiding deadlock), but we also see splitting at finer granularities, and we are told of plausible implementation techniques for both ARM and POWER, which may be used in current implementations, which would lead (sometimes rarely) to such splitting. The architectures explicitly do not guarantee single-copy atomicity for misaligned accesses within cache lines, or indeed commit to any particular cache-line sizes, so programmers should not rely on that and our semantics should not guarantee it. In particular, we should split misaligned accesses into the architectural one-byte units rather than into the two subaccesses lying within distinct cache lines.

2.4 Atomicity of Load Instructions

Similarly to store instructions, when the footprint of a load instruction is not sufficiently aligned, the architectures regard the load as split into atomic single-byte units. The following variant of the message-passing litmus test illustrates this. Thread 0 has two single-byte store instructions to adjacent (and individually trivially aligned) footprints, with an `lwsync` memory barrier to keep them in order as far as other threads are concerned (an `hwsync` or ARM `dmb sy` would be equivalent here). Thread 1 has a single misaligned load-half-word `i(1:2)` that reads both addresses, with its two single-byte read events `e1` and `e0`, preceded by a load-byte `i(1:1)` of the second address with a read event `d`.

In the execution shown, `e0` and `d` read from the second of Thread 0's writes `c`, while `e1` reads from the initial state, ignoring Thread 1's first write `a`. This is observable on POWER and (using `dmb sy` in place of `lwsync`) ARM, illustrating that `e1` can be satisfied early, before `e0` and indeed also before `d` (as otherwise the `lwsync` would have forced `a` to have been propagated to Thread 1 and `e` would have had to read from `c` instead of from the initial state). As for stores, splitting is also observable at various other boundaries.



Test MP+lwsync+misaligned2+127

Test	Archs	POWER 7 h/w	ARMv8 h/w
`MP+lwsync+misaligned2+0(x)`	forbid	0/10G	0/9.6G
`MP+lwsync+misaligned2+1(x)`	allow	0/10G	0/9.6G
`MP+lwsync+misaligned2+3(x)`	allow	0/10G	0/9.6G
`MP+lwsync+misaligned2+7(x)`	allow	0/10G	0/9.6G
`MP+lwsync+misaligned2+15(x)`	allow	0/10G	0/9.6G
`MP+lwsync+misaligned2+31(x)`	allow	26k/10G	0/9.6G
`MP+lwsync+misaligned2+63(x)`	allow	47k/10G	1.9M/9.6G
`MP+lwsync+misaligned2+127(x)`	allow	3.9M/10G	2.9k/9.6G

2.5 Coherence

Many relaxed memory models, including those of the mainstream multiprocessor architectures and C/C++11 atomics, provide some *coherence* guarantee. In the non-mixed-size setting this is abstractly characterised by requiring that in any complete execution, for each abstract location, there is a total coherence order over all writes to that location, with reads from that location (that are themselves ordered in some way, e.g. by program order in the same thread, or by a combination of barriers and dependencies) respecting the coherence order.

In hardware implementations of relaxed-memory architectures, the coherence relationship between two writes may be established relatively late, after (in hardware execution time) they have been committed and after they have been read from. For example, a coherence relationship may be established when one write wins a race to pass a join-point in a storage hierarchy or a race for cache-line ownership. This makes for a delicate interplay between coherence and other ordering constraints, e.g. from memory barriers: coherence cannot always be transitively combined with other ordering relationships. In particular, the POWER lwsync barrier "Group A" of actions before the barrier, is not closed under coherence [36, Z6.3+lwsync+lwsync+addr, §11][33, blw-w-006, §6]). Previous operational models for POWER [33, 34, 35, 38] and ARM [39] follow implementation in this respect, establishing coherence relationships incrementally. The first does so explicitly, maintaining a partial order over writes to the same address that records the coher-

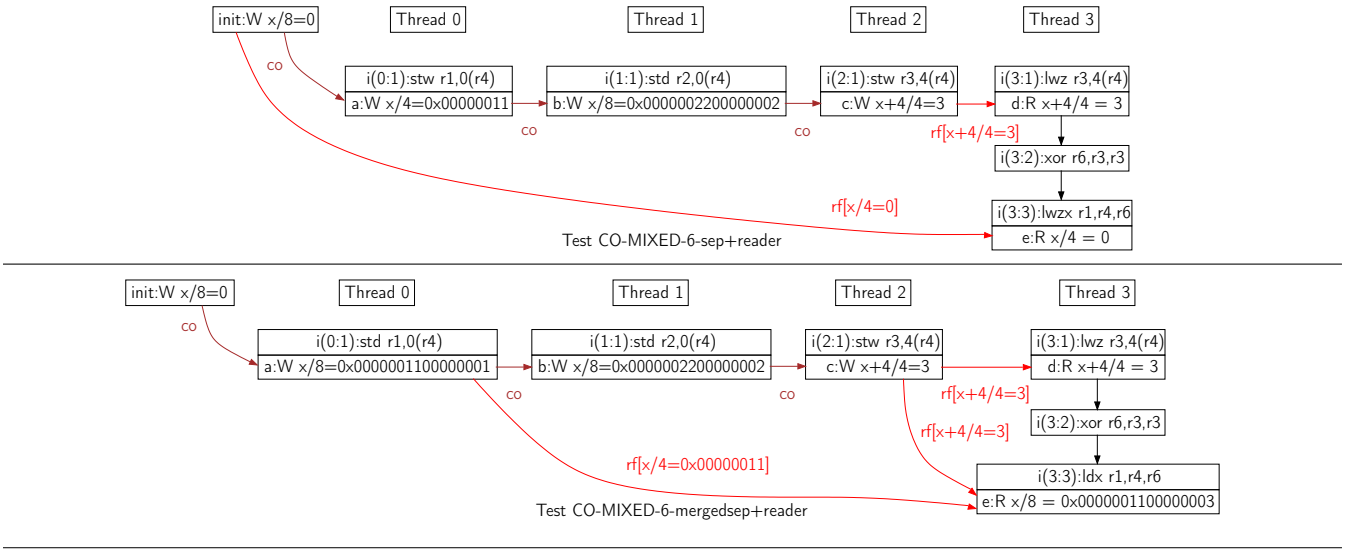
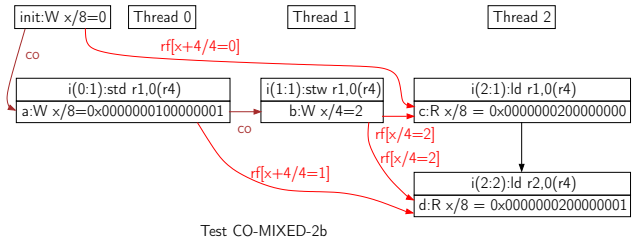


Figure 1.

ence relationships established so far; the second does so implicitly, as writes propagate down a hierarchy.

In the mixed-size setting coherence must be generalised to handle writes that have overlapping but non-identical footprints. Technically, one can think of this in two ways: either as a coherence relation between atomic write events (with perhaps-overlapping footprints), or as per-byte-address coherence orders together with some consistency conditions between them for writes with multiple-address overlaps. The former seems simpler to work with mathematically and is also a better match to microarchitecture, where writes do propagate and win or lose coherence races as atomic units. However, one does need to interpret such coherence edges with care. Consider the execution below



with two writes

```
a:W x/8 = 0x0000000100000001    (* Thread 0 *)
b:W x/4 = 0x00000002........    (* Thread 1 *)
```

related by coherence, with `a` $\xrightarrow{co}$ `b` (as witnessed by executions with a final state of `x/8=0x0000000200000001`).

A read `c` of `x` on a third thread can see just `b`, ignoring `a` even though it is coherence-before `b`, and then another read `d` can see their combination. Microarchitecturally, this can occur in several ways. A simple one is a behaviour in which `c` reads `b` before (in hardware execution time) `a` wins a coherence race with `b`. It is observable on POWER 7 (603/2.2G) but not on these ARMv8 implementations, again unsurprisingly so as they appear to be multi-copy atomic. In general, reading from a write does not guarantee that the effects of other writes, that will eventually end up in the coherence order before the first write, are also visible at the point of that read.

Coherence over mixed-size writes is simplified by the fact that, in the ARM and POWER architectures, as we saw in §2.3, 2.4, store instructions, even if misaligned, generate write events that are each of size 2^n bytes and 2^n-aligned for some natural number n. This rules out sets of writes such as {`a`, `b`, `c`} or {`d`, `e`, `f`}:

```
a: W x  /2     = 0x0302....   d: W x  /2 = 0x0302....
b: W x+1/2     = 0x..1211..   e: W x+1/2 = 0x..1211..
c: W x/1,x+2/1 = 0x23..21..   f: W x+2/2 = 0x....2120
```

We do have to consider sequences of primitive coherence edges, with each edge between two writes with non-empty overlap, but whose endpoint writes do not overlap. However, because writes are each 2^n-aligned and size 2^n bytes for some n, the sub-footprint relation is a tree, so if two footprints overlap then one must be included in the other, and hence in any such sequence there must exist an intermediate write in the sequence that overlaps with both of the endpoints.

That leaves us with examples such as the top of Fig. 1, which has three writes:

```
a: W x  /4 = 0x00000011........    (* Thread 0 *)
b: W x  /8 = 0x0000002200000002    (* Thread 1 *)
c: W x+4/4 = 0x........00000003    (* Thread 2 *)
```

with `a` $\xrightarrow{co}$ `b` $\xrightarrow{co}$ `c` (observable in executions where the final state of `x/8` is `0x0000002200000003`) and where `a` and `c` have disjoint footprints. If coherence is transitively closed (and without that it is hard to use in reasoning) then `a` $\xrightarrow{co}$ `c`. What is the real meaning of such a transitive edge? At first sight one might think it means that another thread (Thread 3) that reads the two subfootprints separately, with a barrier or dependency to ensure local ordering, will never see `c` and fail to see `a`, but on POWER we observe that (7.3k/1.8G).

```
d: R x+4/4 = 0x........00000003    (* Thread 3 *)
<address dependency>
e: R x  /4 = 0x00000000........    (* Thread 3 *)
```

We also observe executions (7.2k/1.8G) in which Thread 3 sees `a` (i.e., not overridden by the coherence-later `b`, even though `b` is a coherence-predecessor of the `c` which it has seen).

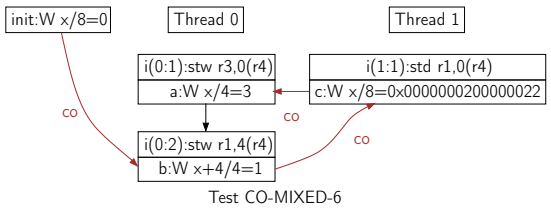
These observations might appear counter-intuitive at first sight, but they have straightforward microarchitectural explanations. Sup-

pose Threads 2 and 3 are close together, sharing one level of cache/store-buffer, then c can reach Thread 3 before being visible to other threads, and before any coherence decisions have been made. It is no surprise that the initial 0 can be read in the other half of the location, by Thread 3 reading from the shared level of cache. Slightly more interesting is the case (not shown) where Thread 3 instead sees a. This could occur if Threads 2 and 3 are close together, as above, and Threads 0 and 1 are not neighbours of each other or of 2+3. As before, c can reach Thread 3 and be read from before reaching other threads, and before any coherence decisions have been made. Then the coherence mechanism between the thread neighbourhoods 0, 1, and 2+3 can settle coherence among {a, b, c}. Write a wins, reaches Thread 3, and is read from, then writes b and c, in that order, take their places in coherence.

Interestingly, one would expect coherence decisions between far-away neighbourhoods to be made at cache-line granularities. This means when write a reaches Thread 3, it has to be locally merged with the partial write c which has not yet taken its place in coherence. This suggests the variation in the bottom of Fig. 1 should also be observable, and indeed it is (6.8k/1.8G).

All these observations require non-multi-copy-atomic behaviour, so again it is unsurprising that we do not see analogous results on the ARMv8 implementations tested.

Another interesting case is below: the simple disjoint write/write thread-local reordering we saw in the second example of §2.2 can also lead to cycles in the union of coherence and program order. This test has two disjoint writes on Thread 0 and a write to the combined footprint on Thread 1; it asks if the former two can be coherence-ordered against program order, with the latter write coherence-between them:



Test CO-MIXED-6

The interesting outcome is with b $\xrightarrow{co}$ c $\xrightarrow{co}$ a, with

```
a: W x  /4 = 0x00000003........     (* Thread 0 *)
b: W x+4/4 = 0x........00000001     (* Thread 0 *)
c: W x  /8 = 0x0000000200000022     (* Thread 1 *)
final  x/8 = 0x0000000300000022
```

This is not observed on POWER 7 or on our ARMv8 implementations. The former is unsurprising, as current POWER implementations appear not to do out-of-order write commitment, and any write propagation effects may well only be at a cache-line granularity. The latter contrasts with other ARM write-write reordering, e.g. MP+po+addr. However, discussion with the vendors confirm that it is architecturally allowed for both, simply because a and b are to disjoint footprints and so are not architecturally locally ordered (despite their program-order relationship). Our models permit it.

2.6 Write Forwarding

In the non-mixed-size context a load can read a value written by a store from the same thread while the store is still speculative (i.e., a branch condition preceding the store has not been resolved yet). This is illustrated by the PPOCA litmus test [33]. Implementation microarchitectures exhibit such behaviour by forwarding uncommitted speculative writes to program-order-later reads. There are several interesting variations of such forwarding in the mixed-size setting (these tests are not shown but they are included, with the others, in the supplementary material).

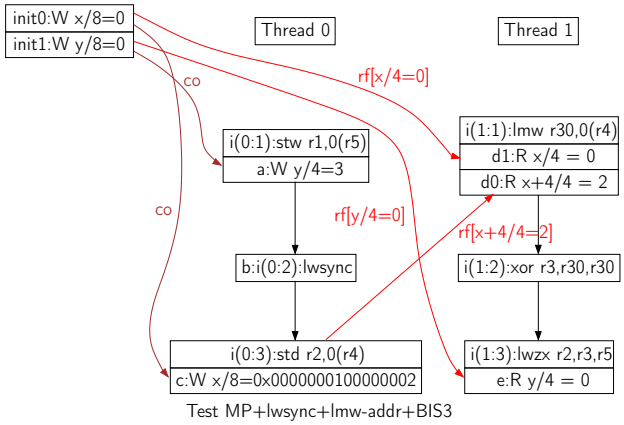
In Test PPOCA-MIXED-3, a slice of write can be forwarded to a narrower load. Test PPOCA-MIXED-1 has a read partially satisfied by forwarding a speculative write, with the rest satisfied from memory. Test PPOCA-MIXED-2 involves a read satisfied by forwarding two writes. All three of these behaviours are allowed by both architectures; they are observed on current hardware as below.

Test	Archs	POWER 7 h/w	ARMv8 h/w
PPOCA-MIXED-3	allow	7/3.4G	69k/6.0G
PPOCA-MIXED-2	allow	0/3.4G	63k/6.0G
PPOCA-MIXED-1	allow	0/3.4G	28k/6.0G

2.7 Load/Store Multiple and Load/Store Pair

The IBM POWER ISA contains *store multiple word* stmw and *load multiple word* lmw instructions, that write or read up to 32 consecutive 4-byte words into the low-order bytes of corresponding registers (in the Server version of the architecture, which is our focus, these instructions are only available in big-endian mode). Even if word-aligned, their writes and reads must be split into 4-byte units, with no ordering amongst them, broadly similar to the splitting in §2.3 and §2.4. We observe this on POWER 7 in tests MP+stmw+addr+124 and MP+std+lmw (not shown, 8.7M/3.4G and 12M/3.4G respectively). The first has Thread 0 comprising a store-multiple of two aligned 4-byte words that cross a cache-line boundary, read by two aligned 4-byte reads on Thread 1 separated by an address dependency. The second has Thread 0 comprising an 8-byte aligned store-doubleword, read by a similarly aligned load-multiple of two words.

However, the fact that lmw reads into multiple registers raises a new question that our models do not currently address. The above shows that the read requests to memory must semantically be split, but then after one read request from a lmw is satisfied, can program-order later instructions that read from the register that takes that result go ahead even before the other read requests from the lmw are satisfied? The test below illustrates this: Thread 0 has an 8-byte write (aligned, and hence single-copy atomic) c to x/8, preceded by an lwsync barrier and a write a to y. Thread 1 has a load-multiple instruction that reads from x/4 into register r30 (d1) and from x+4/4 into register r31 (d0) (zero-extending both values). In the execution shown, the first of those reads (d1) is satisfied first, from the initial state for x/4 rather than from c, which lets an address dependency to the read e of y go ahead and read from the initial state for y. This must be before (in machine execution time) the second read (d0): that does read from write c, and the lwsync means that write a to y must have propagated to Thread 1 before c does, but the read e of y did not see a. This is observable on POWER 7 (4.7k/3.5G).



Test MP+lwsync+lmw-addr+BIS3

The ARMv8 A64 64-bit instruction set includes load/store-pair instructions, for which the same question can be asked (load/store-multiple are part of the ARMv8 A32 32-bit instruction set, which is not covered by this work). The analogue of the first test is observable (`MP+stp+addr+60`, 20M/6.0G), while the analogues of the second and third are not (`MP+str+ldp`, `MP+dmbsy+ldp-addr+BIS3`).

Our models could be adapted to permit this behaviour but at present they do not (it needs a stronger form of intra-instruction parallelism), so these multiple and pair instructions are not in the fragments of the ISAs that we support. Neither ISA has instructions that read from (say) just the top half of a register, so the question does not arise in the remainder of the ISA.

3. Mixed-Size Semantics for Hardware

We now describe our semantic models for Power and ARM that handle the mixed-size phenomena of §2, including some more technical issues that have to be dealt with.

Context The new models extend those developed in previous work [33, 34, 35, 38, 39], which we recall first. These are operational models: at the top level, each defines a type of whole-system machine states, a type of transition labels, and a total computable function that, given a state, calculates the set of all its possible transitions. Each model can thus be executed as a test oracle, to compute (for a small litmus-test program and initial state) the set of all model-allowed final states, by an exhaustive memoised search. This set can then be compared with the sets of final states observed by running the test experimentally on production hardware implementations, using the test harness constructed by the `litmus` tool [44, 28]; any discrepancy between the two sets indicates either a flaw in the model, a flaw in the hardware, or a place where the architecture (and the model) is intentionally looser than the behaviour of that specific hardware implementation. The models can also be executed interactively, letting the user explore a single execution path (backtracking as desired), with command-line and web interfaces to show the state at each point.

The concurrency models are expressed in the Lem language [55] as type and function definitions, from which Lem generates pure OCaml code used in tools. Lem can also generate theorem-prover definitions, for HOL4, Isabelle/HOL, and (to a limited extent) Coq; this should enable mechanised proof about the models in the future. As an experiment towards that, we have recently proved termination for the Isabelle version of most functions (in a slightly earlier version of the model than that of this paper) except the Sail interpreter and the fragment processing.

Each model is factored into three parts:

- the semantics of individual machine instructions in isolation. This is expressed in Sail [38, 39], a language reminiscent of (but cleaner and more strongly typed than) the vendor pseudocode languages used in their architecture texts. We use Sail definitions of substantial fragments of the POWER [38] and ARM [39] ISAs, derived (respectively) semiautomatically and manually from those. The typed Sail AST is deeply embedded in Lem; a Sail interpreter gives an operational semantics that produces primitive memory and register read/write events [38, §2.2].
- the thread semantics, loose enough to admit the observable behaviour of pipeline optimisations, including out-of-order and speculative execution. At any moment each thread may have many instruction instances in flight, each with partially executed instruction semantics, and others that have been committed. Much of the thread semantics is common to the ARM and POWER models.
- the storage subsystem semantics, handling the propagation of writes and barriers (and, for ARM, read requests) between threads. This abstracts from the storage hierarchies, cache protocols, and interconnects of hardware implementations. For POWER this is based on the coherence-by-fiat model of [33, 35]; for ARM there are the low-level (more microarchitectural) Flowing model and the higher-level (abstracting from thread interconnect topology) POP model of [39].

This structure gives the thread and storage subsystem semantics enough of a microarchitectural flavour to let us discuss them in detail with ARM and POWER architects, which is necessary for model design and for model validation, to ensure the models capture the architectural intent (black-box testing, while also necessary, would not suffice alone). At the same time, they are abstract enough not to get bogged down in the hardware implementation detail that is not programmer-observable, which would be too complex to work with. They really are architectural envelope models, capturing, to the best of our knowledge, the envelopes of all behaviour which are intended to be allowed. For the instruction semantics, where there is less ambiguity in the existing architecture texts, and less concurrency-related subtlety, but a bigger mass of detail for these relatively large ISAs, expressing them in a language close to those of the existing informal specifications also helps ensure we correctly capture the intent.

Extending all this to the mixed-size setting required changes to all parts. There is a pervasive change to the basic types for read and write events, which now use footprints (of a concrete address and a size in bytes) rather than just addresses. Below we describe the changes to the instruction and thread semantics, which were broadly common to ARM and POWER, and then the storage subsystem changes for each.

3.1 Instruction and Thread Semantics

For the instruction semantics, our ISA metalanguage, Sail, had to support operations on bitvectors (for register values and operations) and bytevectors (at the interface to the thread semantics of the memory model) throughout. To make the user interface make sense for those familiar with the vendor ISA descriptions, we arranged for the indexing of bitvectors (e.g. for parts of registers) to correspond to the existing conventions: ARM bit indices decrease from most-significant-bit to least-significant-bit, while POWER bit indices increase, and different registers have different starting indices.

The previous thread semantics relied on the Sail semantics of each instruction making at most one memory read or write, which simplifies the semantics of instruction commit; to maintain this, some of the instruction descriptions needed to be rewritten, e.g. to do a single wide write in place of multiple writes, and an intermediate layer was added to split wide or misaligned write events into the correct architecturally atomic units, as in §2.3. Misaligned and wide reads must also be split, as in §2.4, but this was more involved, introducing new intermediate instruction states in which some but not all of the fragments of such a read have been satisfied.

Write forwarding, from an uncommitted write on a speculative path (after an as-yet-unresolved control dependency), introduced additional complications to that (§2.6). Then both register and memory reads need to be able to read from multiple writes, assembling the correct value from the relevant fragments, as in §2.1; a common abstraction of fragments served both.

Finally (for the thread semantics), all the calculations of whether instructions might access the same memory address needed to take footprints into account.

The whole model is currently 12600 non-comment lines of Lem specification, as below, together with 3400 and 3693 lines of Sail for the fragments of the ARMv8 and POWER ISA specifications

covered, and additional OCaml code for parsing, pretty printing, and suchlike.

`machineDefDebug.lem`	29
`machineDefUtils.lem`	72
`machineDefFreshIds.lem`	15
Sail interpreter (7 files)	6100
`machineDefTypes.lem`	726
`machineDefFragments.lem`	289
`machineDefStorageSubsystem.lem` (POWER)	604
`machineDefFlowingStorageSubsystem.lem` (ARM)	642
`machineDefPOPStorageSubsystem.lem` (ARM)	362
`machineDefThreadSubsystem.lem`	2810
`machineDefSystem.lem`	951

We now describe the more semantically interesting changes required to each of the storage subsystem models, in terms of the prose descriptions of their states and transitions from earlier work.

3.2 POWER Coherence-by-Fiat Storage Subsystem Semantics

Coherence and write propagation To reflect the microarchitectural intuition for POWER explored in §2.5, we first adapt the model of [33, 35] by replacing the per-location coherence partial orders (over the writes to that abstract location) by a single partial order over all writes. The previous model did not propagate a write to a thread when any coherence successor had already been propagated there; now we propagate the non-coherence-superseded *slices* of a write: the `events_propagated_to` for each thread (previously a list of the writes and barriers propagated to that thread) now includes, for each write, a list of its slices (sub-parts of its footprint) which actually become visible to that thread. We allow a thread to satisfy a read only from those slices, and we allow write propagation only if there is some non-empty part of the write which is not coherence-subsumed by the slices of other writes already propagated. Previously it was an invariant that the order of the writes in each `events_propagated_to` list coincided with their coherence order, but now that is not the case. Each pair of writes with overlapping footprint in such a list must be coherence-related one way or the other, but the `events_propagated_to` order must match the coherence order only for pairs where the propagated slices have non-empty overlapping footprints. This accommodates (for example) the propagation of the non-coherence-superseded slice of `a` to Thread 2 after `b` has propagated to Thread 2 in test `CO-MIXED-2b` of §2.5.

Accept write request When a new write request from a thread is received by the storage subsystem, the previous semantics updated the coherence relation to make *the new write coherence-after all writes (to the same address) that have previously propagated to this thread or that have reached their coherence point* (see *coherence point* below). We now make the new write w coherence-after all previously propagated writes w' whose complete footprints overlap the new w, irrespective of their propagated slices, as any such propagated write w' for which only the non-propagated slices overlap the new w will be coherence-predecessors of another propagated write (w'') with propagated slices that do overlap the new w.

Partial coherence commitment The coherence-by-fiat storage subsystem semantics [33, 35] abstracts all other ways that coherence commitments can be made incrementally into a single *partial coherence commitment* rule, allowing the storage subsystem to internally add an arbitrary coherence edge (between a pair of writes to the same address that are not yet related by coherence), together with any edges implied by transitivity, if:

(a) there is no cycle in the union of the resulting coherence order and the set of all pairs of writes (w_1, w_2), to any addresses, for which w_1 and w_2 are separated by a barrier in the list of events propagated to the thread of w_2 (in the non-mixed-size setting this can only happen if w_1 is coherence-before w_2); and
(b) there is no new edge to any write that has reached coherence point.

Condition (a), whose real force is for the `lwsync` barrier, abstracts from the microarchitectural fact that coherence choices in implementations are made in a *hierarchical* storage subsystem, and it prevents the model from making coherence choices that will later lead to deadlock.

At first sight one might think this could be left unchanged in the mixed-size setting, but it should be possible for two writes by two different threads, e.g. `a` to `x/8` and `b` to `x/4`, to propagate to a third thread, in order `b` then (the `x+4/4` slice of) `a`, with an `lwsync` by that third thread in between, even though they become coherence-ordered `a` $\xrightarrow{\mathsf{co}}$ `b`. The above Clause (a) would forbid this, so we modify it to require only that there is no cycle in the union of the resulting coherence order and the set of all pairs of writes (w_1, w_2), to any addresses, for which w_1 and w_2 are separated by a barrier (from any thread) in the list of events propagated to the thread of w_2, and for which w_2 is not coherence-before w_1.

Write reaches its coherence point This transition marks when the coherence order up to a write has become completely determined (preventing other writes later becoming coherence-before it). It can remain unchanged, ignoring the propagated-slice information, for the same intuitive reason that the coherence relation remains over writes, not over their slices: the slice information is just a question of superseding visibility in a local buffer or cache, not about when the writes win coherence races at intermediate or final points.

Propagate barrier to another thread In the previous semantics, the storage subsystem could propagate a barrier it has seen to another thread tid if: (1) the barrier has not yet been propagated to that thread; and (2) for each Group A write, that write (or some coherence successor) has already been propagated to that thread. Here the Group A writes for a barrier were the set of writes propagated before the barrier to the thread that performed it.

Now we need Group A to include the data identifying the slice(s) of each write that propagated before the barrier to the thread tid' that performed the barrier, and we check for each such write that, for each byte of the propagated slice, either that byte of it or the corresponding byte of some coherence-successor write has propagated to the barrier propagation target thread tid.

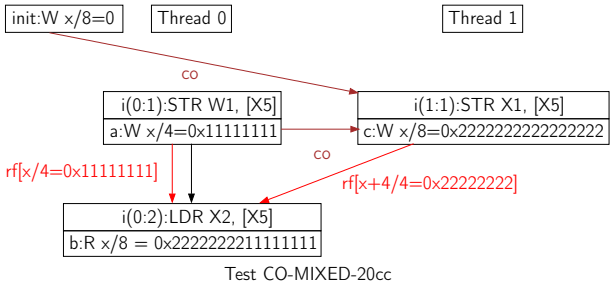
Note that if a proper slice of a write has been propagated to a thread, then coherence-successors of the remainder of the write must already have been propagated there. One might think that we therefore do not need to consider the slices that have been propagated to this thread individually — that it would be enough to check that slices that coherence-cover this complete write have been propagated to the barrier propagation target. But some of the coherence-covering writes of the different slices could be coherence-between the coherence-covered writes, so it seems one should consider the slices propagated to this thread individually, and for each check that coherence-covering-successor-slices have been propagated to the target thread.

3.3 ARM Flowing and POP Storage Subsystem Semantics

Adapting the Flowing model In contrast to the POWER coherence-by-fiat model, the Flowing model of [39] does not have an explicit coherence relation. Instead, Flowing maintains a concrete hierarchy of queues above a byte-array memory. Read, write and barrier requests are received from the threads and enter the top

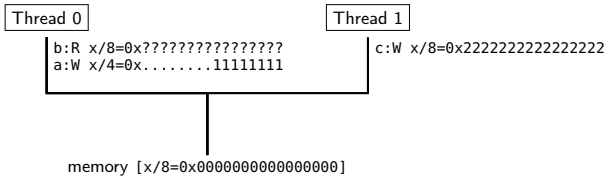
of the associated queue. Adjacent requests in a queue can swap positions subject to a *reorder condition*. Requests at the bottom of a queue can be removed from the queue and placed at the top of the next queue in the hierarchy. When a write request is removed from the root queue the memory is updated with its value. A read request can be satisfied when it is adjacent to a write request to the same address or when it is removed from the root queue (using the value stored in memory).

To accommodate the behaviours explored in §2, we first have to change the satisfy-read transitions. As a read request can now be satisfied from multiple writes, the *Satisfy read from segment* transition has to allow partial satisfaction of the read, and the state has to account for reads that are partially satisfied. The state of the new model records, for each read, the slices of its footprint that have not been satisfied yet, together with the slices that have been satisfied and the writes that satisfied them. The transition is now enabled if the footprint of a write overlaps the unsatisfied slices of the read. When the transition is taken, the state is first updated to record the write has satisfied the overlapping unsatisfied slices of the read. We then have to consider two cases: in the first case the read has no more unsatisfied slices, in which case a read response is sent to the thread subsystem that issued the read, together with the writes that satisfied it, and the read is removed from the storage subsystem. In the second case the read still has unsatisfied slices. In this case we have to take care not to break single-copy atomicity, as we will explain using the test below.

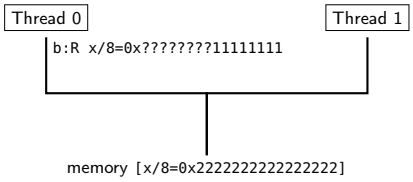


Test CO-MIXED-20cc

Consider the following intermediate Flowing state, reached by committing the write a, issuing the read b and committing the write c:

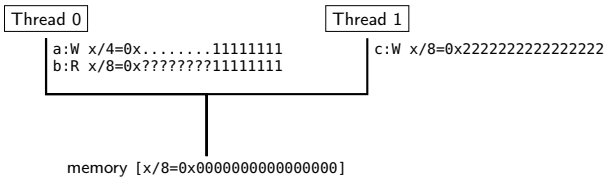


In this state we can partially satisfy b with a. If the model did not take any measures to guarantee single-copy atomicity, we could continue by flowing a to memory and then flowing c to memory, reaching the following state:



At this point b could flow down and satisfy its unsatisfied slices with the value in memory, resulting in the combined read value of 0x2222222211111111. As the final value in memory implies the coherence order a $\xrightarrow{co}$ c, that would be a violation of single-copy atomicity. To prevent this, the transition where a write partially satisfies a read also swaps the position of the write and the read in the queue. In addition, to make sure the read and the write remain in this order, we add to the reorder condition that a write that satisfied a read can never again be reordered with it.

Going back to the example above, after b is partially satisfied by a we reach the state:



(notice that a and b swapped position) which guarantees that if the remaining slices of b are to be satisfied by c the coherence order of a and c will be c $\xrightarrow{co}$ a.

Adapting the *Satisfy read from memory* rule is more straightforward as the transition fully satisfy the unsatisfied slices of the read. It involves minor changes to account for the fact that a read might already be partially satisfied.

Finally we adapt the reorder condition. Where before two memory access requests could not be reordered if they were to the same address, we now check whether there is an overlap, taking footprints of writes and unsatisfied slices for reads.

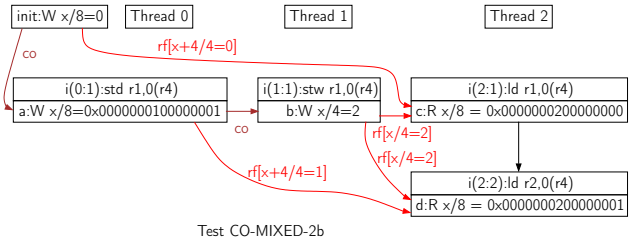
Adapting the POP model The POP model of [39] replaces the hierarchical queue structure with a more abstract explicit partial order between requests (*order-constraint*). This model makes use of the Flowing *reorder-condition* to determine how *order-constraint* should evolve when taking an *Accept request* or *Propagate request to another thread* transition. The modifications to the *reorder-condition* we described above are the same here and the only thing remaining to be adapted is the *Send read-response to thread* transition. This follows the same lines as the adaptation of the Flowing *Satisfy read from segment* transition. Where in the new Flowing model, when a read is partially satisfied by a write, we swap the positions of the read and the write in the queue, in the POP model we simply swap the positions of the read and the write in the *order-constraint*. This is enough to guarantee the single-copy atomicity required by the ARM architecture.

4. Sequential Consistency for Mixed-Size?

Barriers do not recover SC for mixed-size ARM or POWER A standard result for relaxed memory models, and a property that architectures have normally been thought to intend and to guarantee, is that inserting enough barriers in a concurrent program restores sequentially consistent behaviour. The only exception that we were previously aware of is Itanium.[2] Perhaps surprisingly, in the mixed-size setting neither ARM nor POWER have this property, as the example below from §2.5 shows: there is no way to totally order these four events with each read reading each byte from the most recent write to that byte. This execution is architecturally allowed on both ARM and POWER and observable on current POWER implementations (one would not expect it to be observable on current ARM implementations as we have not ob-

[2] The Intel Itanium specification [22] defines a non-multi-copy-atomic model where the strongest barrier is not sufficient to regain multi-copy atomicity, for normal accesses, and hence insufficient to regain SC for them; regaining SC requires the Itanium store-release and load-acquire instructions. It is unclear whether Itanium implementations have actually exploited that weakness, but accommodating it led to the weak semantics of the C/C++11 SC fence [56, 46, 34, 57].

served other non-multi-copy-atomic behaviour). Adding a barrier between these two reads makes no difference in the models, and the result remains observable with a `sync` barrier on POWER 7 (Test `CO-MIXED-2b-sync`, 48k/2.2G).

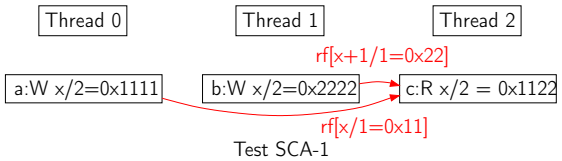


Test CO-MIXED-2b

Characterising the behaviour of fully barriered programs We therefore characterise what guarantees these architectures do give when inserting strong barriers (`sync` or `dmb sy`) between any two instructions in program order. For conciseness, in the following we will call these programs "fully barriered". For simplicity we restrict to the case of programs that have no misaligned memory accesses, which would also need the store and load splitting of §2.3, §2.4.

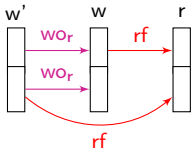
As a first attempt at an axiomatic characterisation of hardware behaviour for fully barriered mixed-size programs (without misaligned accesses) consider the following, henceforth called Bytewise SC (BSC). Partition all read events and write events into *subevents* (also *subreads* and *subwrites*) of the smallest size supported by the architecture (for POWER and ARM this is one byte), and record which subevents were generated by the same event in an irreflexive, symmetric relation si. Now define a candidate execution to consist of the subevents with the usual relations po, rf, and co — but per byte, and with po lifting program order to a relation on the subevents. We require that coherence be compatible with si in the following sense: $w_i \xrightarrow{\mathsf{co}} v_j \implies w_{i'} \xrightarrow{\mathsf{co}} v_{j'}$ whenever $\{(w_i, w_{i'}), (v_j, v_{j'})\} \subseteq \mathsf{si}$ and $w_{i'}$ and $v_{j'}$ have the same address. We then call a candidate execution BSC if there is a total order on the *subevents* that agrees with po, co, and rf (a subread having the value of the most recent preceding subwrite in the order).

This can be shown to admit all behaviour of fully barriered mixed-size POWER and ARM programs without misaligned accesses. For example, the behaviour above is witnessed by the subevent order $\mathtt{c}_7 \to \mathtt{a}_7 \to \mathtt{a}_3 \xrightarrow{\mathsf{co}} \mathtt{b}_3 \xrightarrow{\mathsf{rf}} \mathtt{c}_3$, suitably extended for the other subevents. (x_i denotes the i-th byte-sized subevent of an event x, e.g. $\mathtt{c}_3$ is the subwrite `0x02` of `c`.) However, it gives too weak a guarantee to be suitable for programming and is weaker than hardware for fully barriered programs; for example, the undesirable behaviour of the test below is allowed in BSC. Here the read c is satisfied from a mix of the writes `a` and `b`, while in ARM and POWER one would want it to be satisfied completely by a single write, `a` or `b`, whichever wins the race.



Test SCA-1

This execution violates the principle of single-copy atomicity: for any read r there must be a total order wo_r over the writes it reads from such that each subread of r reads from the wo_r-maximal subwrite. This can be more



formally defined as follows: an execution is single-copy atomic if for each read r there exists a total order over all same-address subwrites wo_r compatible with si and such that there are no cycles of the form $\mathsf{rf}_r; \mathsf{si}; \mathsf{rf}_r^{-1}; \mathsf{wo}_r^{+}; \mathsf{si}$, where rf_r is rf restricted to r's subreads.

The above definition allows the ordering of writes wo_r to be different for each read event r. In most cases, including POWER and ARM, there is a notion of coherence that requires a global ordering of overlapping writes. In these cases we can specialise single-copy atomicity to the following, where wo_r always coincides with coherence: $\mathsf{rf}; \mathsf{si}; \mathsf{rf}^{-1}; \mathsf{co}^{+}; \mathsf{si}$ must be acyclic (c.f. [42, B2.6.2]). We now define BSC+SCA as BSC with the latter single-copy atomicity axiom added.

Theorem 1. *All behaviours that fully barriered POWER and ARM programs with no misaligned accesses exhibit are allowed by BSC+SCA.*

The proof, in the supplementary material, takes an arbitrary trace *tr* of the POWER or ARM model, and constructs a total order on the byte-sized subevents that matches program order, coherence, and reads-from (and from-reads) of the trace. For ARM the proof uses a lemma that states that any two writes that are related by paths of coherence, reads-from, from-reads, and program-order edges are already related in the same way in order-constraints of the final state of *tr*. For POWER the key result is that for any path in the graph of events with coherence, reads-from, from-reads, and program-order edges that ends with an edge (e, e'), in the state in *tr* when e' is accepted into the storage subsystem all reads on the path have been satisfied and all writes from the path have been propagated to all threads.

Recovering SC on ARM If all memory events have the same size, any complete execution totally orders same-address events (except for read-read pairs) in terms of the relations coherence, reads-from, and from-reads. In the example of `CO-MIXED-2b`, however, there is not such a total order: the read `c` observes a state where `a` and `b` are not ordered yet. What is necessary to prevent the behaviour of `CO-MIXED-2b` is multi-copy atomicity: `c` must not be satisfied before `b` is visible to all threads and thus ordered with `a`. In ARM this is exactly the behaviour that acquire reads in combination with release writes provide: replacing `b` with a write-release and `c` with a read-acquire in the test forbids the non-SC behaviour, because `c` can only be satisfied from `b` when it is propagated to all threads, at which point `a` and `b` are ordered: either `a` is ordered before `b` and `c` returns `0x0000000200000001`, or `b` is ordered before `a`, but then it is $\mathtt{b} \xrightarrow{\mathsf{co}} \mathtt{a}$. This gives an intuition for the following theorem.

Theorem 2. *An ARM program whose only reads are acquire reads and whose only writes are release writes and that has no misaligned memory accesses has sequentially consistent behaviour.*

The intuition behind this is that if all memory accesses are release/acquire accesses, the thread semantics is forced to behave sequentially, the storage subsystem keeps all release/acquire events in the order they were accepted into storage, and multi-copy atomicity ensures that the reads-from relation agrees with some total order on same-address events.

The proof, in the supplementary material, constructs a total order on the reads and writes of a given POP trace that matches program order, coherence, and reads-from (and from-reads). The key point of the proof is that at the point when a read is partially satisfied it has to be fully propagated, and therefore all writes it will read from fully propagated and totally ordered by order-constraints.

Implications for Java and C memory models Both Java and C/C++11 language-level models guarantee sequential consistency in particular circumstances, for volatiles and for SC atomics re-

spectively; one should therefore ask whether the above observation invalidates the usual compilation schemes. Fortunately it does not: neither language permits mixed-size accesses of those kinds. Low-level systems code does exploit them, however, as in the examples mentioned in §1.

5. C/C++11 Mixed-Size

In this section we extend the formal C/C++11 axiomatic model of Batty et al. [46, 45, 56] to support mixed-size nonatomic accesses. For brevity we describe the changes to that model in prose; we refer the reader to [45, 46, 57] for an introduction to C/C++11 concurrency, and to the supplementary material for the full mathematical definition of the extended model.

We first add *footprints* to read and write events, replacing the previous addresses. The type of footprint is abstract, to support later integration with a variety of C memory layout models; in a concrete memory layout model it could be implemented as pairs of a concrete address and a size in bytes, as in the hardware models. Footprints are manipulated only with is_empty and $inclusion$ tests and with $empty$, $difference$, $intersection$, and $bigunion$ operations.

Reads can now read from multiple writes, so to make explicit which part the read is reading from which write we also add footprints to rf-edges. In the C/C++11 concurrency model coherence and the SC order are only over accesses to atomic locations. In the ISO C standard mixed-size overlapping atomic accesses are forbidden by the effective type rules, and the general combination of atomic and non-atomic accesses (e.g. `char *` accesses to atomic locations) is an open problem [52] which we do not consider here. All the accesses to each atomic location will therefore be of the same size, and so there is no need to add footprint information to coherence or to the SC order.

Then we add footprints to the *visible side effects* relation vse, where $(w, r) \in vse$ means that the write w is visible to the read r. Visible side effects are a C/C++ notion; the relation does not have an equivalent in hardware models. In the original C/C++11 model the visible side effects to a read r are all the writes w to the same location as r for which w happens before r, but where there is no write to the same location that happens between w and r ("happens before" or hb is a partial order calculated from the relations arising synchronising actions and program order). In the mixed-size model it is possible that only a part of a write is visible to a read. The footprint of the vse-relation denotes which part is visible, and it is defined as follows: if a write w happens before r and there is a non-empty part f of the footprint of w that is not overwritten by writes hb-between w and r, then (w, f, r) is a visible side effect.

Finally, we adapt the following *consistency* and *race* predicates of the original model that concern non-atomics.

Well formed rf To support reading parts of multiple writes we made the following changes to the consistency predicate *well-formed-rf*.

- The original predicate requires that a read can read from at most one write. In the mixed-size model this is only required for atomics; for non-atomics we require that there is at most one rf-edge between every pair (w, r) of write and read, and that the footprints of all the rf-edges to the same read are disjoint.
- The original predicate requires for all $(w, r) \in \mathsf{rf}$ that w and r are to the same location. Now we require instead that the footprint f of $(w, f, r) \in \mathsf{rf}$ is non-empty, and included in both the footprint of w and in the footprint of r. Note that we do not require that the union of the footprints of all the rf-edges to r equals the footprint of r. The reason is that it should be possible to have a partly indeterminate read (which means the execution is undefined).
- The original predicate requires for all $(w, r) \in \mathsf{rf}$ that the value read by r equals the value written by w. Since a read can now read from multiple writes we have to combine the (parts of) the values of the writes. To determine this value we use a function $combine\_cvalues$ whose implementation is left to users of the model. It takes a set of tuples (v, f_1, f_2), where v is the value of a write, f_1 the footprint of the write and f_2 the footprint of the rf-edge from that write. The return value is an option type, either *nothing*, e.g. in the case that the set is empty, or the constructed value.

Consistent non-atomic rf The original *consistent-non-atomic-rf* predicate requires that non-atomic reads only read from visible side effects. Both rf- and vse-edges now have footprints, but it would be wrong to require that every rf-edge (w, f, r) is in vse: in a racy program there could be distinct writes w and w' with the same footprint f such that (w, f, r) and (w', f, r) are both visible side effects, and r could read only a part f_1 of w and read the rest f_2 from w'. This means that the rf-edges (w, f_1, r) and (w', f_2, r) are not in vse. Although the execution is racy, it should be consistent (otherwise the race might not be detected) so the new *consistent-non-atomic-rf* predicate requires that for every rf-edge (w, f, r) there is a vse-edge (w, f', r) such that f is included in f'.

Determinate reads The consistency predicate *determinate-reads* governs whether a load r should read from somewhere or not: the original predicate requires that r has an rf-edge to it if and only if there exists a visible side effect to r. Because in our mixed-size model a read can be partly determinate and partly indeterminate, we instead require that the union of the footprints of the rf-edges to r equals the union of the footprints of the vse-edges to r.

Indeterminate reads The predicate *indeterminate-reads* is a *race* predicate: it does not impose any requirements on executions, but if it is true for some consistent execution it means that the program has undefined behaviour. The original predicate is true if there exists a read that has no rf-edge to it. In our proposed model *indeterminate-reads* is true if there exists a read r whose footprint is not completely covered by the footprints of the rf-edges to r.

Races The original race predicate *data-races* is true if there exist two distinct actions, at least one non-atomic and at least one a write, which are to the same location, from different threads and that are not happens-before related. The predicate *unsequenced-races* is the same but for actions *within* a thread. In our proposed model we no longer require that the actions are to the same location, but instead that they have overlapping footprints (the intersection of the footprints is non-empty).

Looking at the example hardware executions from §2 that could arise from C/C++11 executions, regarding all reads and writes as nonatomic C/C++11 accesses, the first (`MIXED-SEQ-1`) is consistent and defined. The other executions are all inconsistent, since non-atomics can only read from hb-before writes, but each test has another execution which is consistent and racy, with all reads reading from the initial write(s), and so they are deemed to have undefined behaviour in the model, as one would wish.

6. Mixed-Size C/C++11 to POWER

We now sketch an argument to show that mixed-size phenomena introduce no further complication to correctness proof for the standard compilation scheme [58] from C/C++11 concurrency to POWER, by adapting a previous proof attempt [34] to cover the models of §5 and §3. Note, however, that that previous result and proof are now known to be unsound, for unrelated reasons [47, 48, 49]. Specifically, the previous result does not hold for mixtures of SC and non-SC atomic C/C++11 accesses, where

the requirement that the SC order of C/C++11 is consistent with happens-before is not be satisfied in all cases. The C/C++11 axioms for SC accesses need to be fixed to resolve this problem. It is currently unclear what the best fix is, but we expect it to be independent of mixed-size phenomena, and hence the following argument to still apply.

For any given C program p with mixed-size non-atomic accesses, we begin by converting it to another C program p' which has non-overlapping non-atomic accesses (this can always be done by splitting non-atomic accesses to byte-width accesses). Under this transformation there is a natural correspondence between the consistent executions of p and those of p'. In particular, if the original program p is data-race free, so is the transformed program p'. Furthermore there is an isomorphism between the POWER model traces generated for p and a subset of the POWER model traces generated for p'. We conjecture that any POWER trace of the data-race free p' will have a consistent execution in the model of §5. We complete our proof sketch by defining a mapping which converts a given consistent execution of p' to a consistent execution of p. We outline the steps below; more details are available in supplementary material.

For the following, let p be a data-race free C program with mixed-size non-atomic accesses.

Splitting non-atomic accesses We replace all non-atomic accesses of p with a sequence of byte-sized accesses covering the footprint of the former. Let split denote the syntactic transformer which replaces all non-atomic accesses with a sequence of its associated byte-sized accesses. We use the expression $s' \in \mathsf{split}(s)$ if s' is one of the byte-sized accesses obtained by applying split to s. Let p' be the C program obtained by applying split to p.

Correspondence between p and p' executions Observe that split preserves data, control and address dependencies as well as the program order. More concretely, whenever s_1 and s_2 are in some dependency relation or the former is sequenced-before the latter, then for any $s_1' \in \mathsf{split}(s_1)$ and $s_2' \in \mathsf{split}(s_2)$, the same relation holds. Let e be a consistent execution of p. We construct the corresponding consistent execution e' of p' as follows. The action set of e' is obtained by applying split to all the actions in e. Since mixed-size is only allowed for non-atomic accesses, the relations rf, sw, sc, mo, hb of e over atomic accesses can be used directly on e'. The rf relation for non-atomic accesses is obtained from rf of e as follows. Let (w, f, r) be an rf-edge in e. Then, we add an rf-edge from all byte-sized writes corresponding to f in $\mathsf{split}(w)$ to all byte-sized reads corresponding to f in $\mathsf{split}(r)$. The converse of obtaining a consistent execution e of p from a consistent execution e' of p' follows the same reasoning.

Good traces of p' and simulation equivalence A *good* trace of p' is a POWER trace in which all transitions corresponding to a split statement are consecutive. For instance, if a write w is propagating to some thread t and w is due to a split non-atomic write w_o, then all $w' \in \mathsf{split}(w_o)$ propagate to t before other transitions are taken. Observe that in certain cases, some of those transitions will not be allowed by the POWER model. For instance, if a coherence-after write of some w' has already propagated to t, then the particular transition of propagating w' to t is not allowed. However, we do have simulation equivalence between the good traces of p' and the traces of p. Let τ (resp. τ') be a POWER trace corresponding to p (resp. p'). Let two states of the POWER model be observationally equivalent if their thread subsystems are in the same state and for all threads and locations a read returns the same value. We show by induction that if τ is at state s_1 and a transition l is allowed that will lead to s_2, then τ' is at some state s_1' equivalent to s_1, there is some sequence of transitions that end at some state s_2' equivalent s_2. Similarly by induction we show that for any good trace τ' of p', there is some trace τ of p such that both traces end at equivalent states provided that they start from equivalent states.

Constructing a consistent execution for p Suppose that the compilation scheme for the single-sized case were proved correct, and thus correct for p'. Since we assumed that p is data-race free, so is p'. This follows from the observation that the inter-thread part of the hb relation of p' is almost the corresponding relation of p (up to the grouping of split accesses). Then because p' is data-race free, each of its POWER traces has a corresponding consistent execution. By the above simulation equivalence, we could construct a consistent execution of p.

7. Tools and Tests

As mentioned in §3, we compile our models into a tool that allows interactive and exhaustive exploration of small programs, building on earlier work [33, 38, 39]. This combines executable code from the model with front-end and user-interface code; it supports assembly litmus tests and small ELF object files. Extending the tool to support mixed-size required changes throughout. To make interactive exploration usable for complex mixed-size tests, we built a new interface that dynamically displays the current model state in the form of the diagrams used in this paper, augmented with the enabled transitions (for the user to select from) attached to each event or instruction. We use the exhaustive exploration to compare the models to production hardware implementation behaviour, using the `litmus` tool [44] (which we have also extended to support mixed-size tests), to run tests on hardware: a POWER 7 server and five ARMv8-architecture implementations.

- IBM POWER 730 server, POWER 7 CPU, 48 hardware threads
- LG H955 phone, Qualcomm Snapdragon810 SoC, ARM Cortex-A57/A53 CPU, quad+quad core (using the A53 cores)
- iPad Air 2, Apple A8X SoC/CPU, three-core
- Google Nexus 9 tablet, Nvidia Tegra K1 SoC, Nvidia Denver CPU, dual-core
- Open-Q 820 development kit, Qualcomm Snapdragon 820 SoC, Qualcomm Krait CPU, 4-core
- ODROID-C2 development board, Amlogic S905 SoC, ARM Cortex-A53 CPU, quad-core

Our tests include mixed-size handwritten tests, including those of §2 and §3, mixed-size systematically generated tests, and non-mixed-size regression tests.

Systematically-generated tests: These tests are produced by the `diy` test generator [59] from cycles of candidate relaxations, a concise and precise mean to describe violations of sequential consistency. Briefly, a candidate relaxation is an edge from one memory access to another that specifies various conditions such as a dependency from the first access to the second, or that the second access is a read that reads from the first. We have enriched the vocabulary of candidate relaxations by adding decorations that specify the size (byte, half-word, word, quadword) and the offset of memory accesses.

We have generated in this way 2308 ARM and 2460 POWER mixed-size litmus tests. Our tool, in exhaustive mode, was able to terminate (with 2 hours time limit) on 548 ARM litmus test using the Flowing model (2 hours time limit), 565 ARM litmus tests using the POP model (2 hours time limit), and 905 POWER litmus tests (4GB space limit). Experience with slightly earlier versions of the models shows that increasing these numbers should be straightforward with more computation time. For all of these

tests, our models are sound with respect to the hardware mentioned above (except for known errata in the hardware).

Regression tests In addition to the mixed-size tests we have also used a suite of 1407 ARM non-mixed-size litmus tests and 1719 POWER non-mixed-size litmus tests from a library developed in previous work [30, 37, 33, 35], to validate the non-mixed-size behaviour of the models. For all of these, our models are sound with respect to the hardware mentioned above (again except for known errata in the hardware).

8. Conclusion

Our work on ARM and POWER concurrency semantics here brings those models to the point where they cover enough of the architectures to describe the behaviour of real concurrent algorithm implementations, not just litmus tests; they can now be used as a basis for research on reasoning techniques and tools for such.

The models build executions incrementally (as operational models normally do), and so in principle they also support pseudo-random execution, to explore longer paths of larger programs, and thereby support testing of concurrent algorithm implementations *against the architectures*, not just against particular implementations. To make that feasible in practice requires additional performance-oriented engineering to produce semantics-based emulators that exhibit the full envelope of architecturally allowed behaviour; our focus to date has rather been on expressing the semantics as clearly as possible.

Further work on coverage remains: for user code, the models do not support vector and floating-point instructions (these are mostly ISA concerns, with few interactions with the concurrency semantics), or the load-multiple and load-pair issue of §2.7. Completeness for systems code requires much more: exceptions and interrupts, address translation and TLBs, instruction cache behaviour, and other systems-mode instructions.

Semantically, it would be desirable also to have a more abstract presentation, e.g. as a provably equivalent axiomatic semantics. And, while our hardware semantics are broadly compositional in hardware implementation structure, and construct executions incrementally (unlike axiomatic models), they are whole-program semantics; not compositional in program structure. That is an open problem for relaxed-memory concurrency in general, with early steps provided e.g. by the library abstraction work of Batty et al. [60], the program logic of Turon et al. [61] (both for C/C++11), and the program logic of Bornat et al. [62] (for POWER).

At the C/C++11 language level, we have extended the previous C/C++11 axiomatic model to cover non-racy mixed-size accesses, but one would like a solid compilation scheme result to provide assurance about both this and the hardware models, and supporting mixed-size atomics and mixtures of atomic and non-atomic accesses represents another open problem for the design of C/C++11 models.

Acknowledgments

We thank Will Deacon, Richard Grisenthwaite, and Derek Williams for extensive discussions about the ARM and IBM POWER architectures; John Baldwin, Paul McKenney, and Robert Watson for discussions of mixed-size usage in FreeBSD and Linux; and the anonymous referees. This work was partly funded by the EPSRC Programme Grant *REMS: Rigorous Engineering for Mainstream Systems*, EP/K008528/1, EPSRC grant *C3: Scalable & Verified Shared Memory via Consistency-directed Cache Coherence* EP/M027317/1 (Sarkar), an ARM iCASE award (Pulte), a Gates Cambridge Scholarship (Nienhuis). and ANR grant WMC (ANR-11-JS02-011, Maranget).

References

[1] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.

[2] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.*, 27(12):1112–1118, December 1978.

[3] William W. Collier. Principles of architecture for systems of parallel processes. Technical Report TR 00.3100, IBM Poughkeepsie, 1981.

[4] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Memory access buffering in multiprocessors. In *Proc. ISCA '86*, pages 434–442, 1986.

[5] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Program. Lang. Syst.*, 8(1):142–153, 1986.

[6] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988.

[7] James R. Goodman. Cache consistency and sequential consistency. Technical Report Technical Report 61, IEEE Scalable Coherent Interface (SCI) Working Group, March 1989.

[8] Sarita V. Adve and Mark D. Hill. Weak ordering — a new definition. In *Proc. ISCA '90*, pages 2–14. ACM, 1990.

[9] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. ISCA '90*, pages 15–26. ACM, 1990.

[10] William W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Inc., 1992.

[11] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. *Formal Specification of Memory Models*, pages 25–41. Springer US, 1992.

[12] Prince Kohli, Gil Neiger, and Mustaque Ahamad. A characterization of scalable shared memories. In *ICPP: International Conference on Parallel Processing*, pages 332–335, 1993.

[13] F. Corella, J. M. Stone, and C. M. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report RC18638, IBM, 1993.

[14] David L Dill, Seungjoon Park, and Andreas G. Nowatzyk. Formal specification of abstract memory models. In *Proceedings of the 1993 Symposium on Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[15] *The SPARC Architecture Manual, Version 9*. SPARC Int., Inc., 1994.

[16] Hagit Attiya and Roy Friedman. Programming DEC-Alpha based multiprocessors the easy way (extended abstract). In *Proc. SPAA*, pages 157–166, New York, NY, USA, 1994. ACM.

[17] José M. Bernabéu-Aubán and Vicente Cholvi-juan. Formalizing memory coherency models. *Journal of Computing and Information*, 1:653–672, 1994.

[18] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. *WRL Research Report*, 95(9), 1995.

[19] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[20] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Weak memory consistency models. Part I: Definitions and comparisons. Technical report, Department of Computer Science, University of Calgary, 1998.

[21] Prosenjit Chatterjee and Ganesh Gopalakrishnan. Towards a formal model of shared memory consistency for Intel Itanium$^{\text{tm}}$. In *19th International Conference on Computer Design (ICCD 2001), September 2001, Austin, TX, USA*, pages 515–518, 2001.

[22] Intel. A formal specification of Intel Itanium processor family memory ordering, 2002. `http://download.intel.com/design/Itanium/Downloads/25142901.pdf`.

[23] A. Adir, H. Attiya, and G. Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):502–515, 2003.

[24] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, New Mexico, USA*, 2004.

[25] Lisa Higham, LillAnne Jackson, and Jalal Kawash. Programmer-centric conditions for Itanium memory consistency. In *Proceedings of the 8th International Conference on Distributed Computing and Networking*, ICDCN'06, pages 58–69. Springer-Verlag, 2006.

[26] Arvind Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. In *Proc. ISCA '06*, pages 29–40. IEEE Computer Society, 2006.

[27] N. Chong and S. Ishtiaq. Reasoning about the ARM weakly consistent memory model. In *MSPC*, 2008.

[28] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL 2009*, pages 379–391, January 2009.

[29] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP 2009*, January 2009.

[30] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.

[31] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674*, pages 391–407, 2009.

[32] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. (Research Highlights).

[33] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proc. PLDI '11*, pages 175–186, 2011.

[34] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Proc. POPL 2012*, pages 509–520, 2012.

[35] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *Proceedings of PLDI 2012, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing)*, pages 311–322, 2012.

[36] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. Draft available from `http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`, 2012.

[37] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM TOPLAS*, 36(2):7:1–7:74, July 2014.

[38] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proc. MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2015.

[39] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of POPL: the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[40] Sizhuo Zhang, Arvind, and Muralidaran Vijayaraghavan. Taming weak memory models. *CoRR*, abs/1606.05416, 2016.

[41] Linux kernel lockrefs. `https://lwn.net/Articles/565734/`, `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/lib/lockref.c`, `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/include/linux/lockref.h`.

[42] ARM Ltd. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*, 2015. ARM DDI 0487A.h (ID092915).

[43] *Power ISA^{TM} Version 2.07*. IBM, 2013.

[44] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In *Proceedings of TACAS 2011*, pages 41–44. Springer-Verlag, 2011.

[45] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.

[46] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

[47] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. *CoRR*, abs/1611.01507, 2016.

[48] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. Note, available at `http://plv.mpi-sws.org/scfix/`, 2016.

[49] Susmit Sarkar and Peter Sewell. Corrigendum: C/C++11 to POWER concurrency compilation scheme correctness proof. Note, available at `http://www.cl.cam.ac.uk/users/pes20/cppppc/corrigendum.html`, December 2016.

[50] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP*, 2007.

[51] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.

[52] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *Proceedings of ESOP 2015*, 2015.

[53] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of POPL*, 2016.

[54] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Supplementary material. `http://www.cl.cam.ac.uk/~pes20/popl17/`, `http://dx.doi.org/10.17863/CAM.6236`, 2016.

[55] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 175–188, 2014.

[56] P. Becker, editor. *Programming Languages — C++*. 2011. ISO/IEC 14882:2011. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`.

[57] Mark John Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge Computer Laboratory, 2014.

[58] P. E. McKenney and R. Silvera. Example POWER implementation for C/C++ memory model. `http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2011.03.04a.html`, 2011.

[59] Jade Alglave and Luc Maranget. The `diy` tool. `http://diy.inria.fr/`.

[60] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In *Proc. POPL '13*, pages 235–248. ACM, 2013.

[61] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proc. OOPSLA '14*, 2014.

[62] Richard Bornat, Jade Alglave, and Matthew J. Parkinson. New lace and arsenic: adventures in weak memory with a program logic. *CoRR*, abs/1512.01416, 2015.